Comparing Tail Duplication with Compensation Code in Single Path Global Instruction Scheduling*

David Gregg

Institut für Computersprachen,
Technische Universität Wien,
Argentinierstr. 8, A-1040 Wien,
E-mail: dave@complang.tuwien.ac.at
Fax: (+431) 58801-18598

Abstract. Global instruction scheduling allows operations to move across basic block boundaries to create tighter schedules. When operations move above control flow joins, some code duplication is generally necessary to preserve semantics. Tail duplication and compensation code are approaches to duplicating the necessary code, used by Superblock Scheduling and Trace Scheduling respectively. Compensation code needs much more engineering effort, but offers the possibility of less code growth. We implemented both algorithms to evaluate whether the extra effort is worthwhile. Experimental results show that trace scheduling does not always create less code growth and often creates more.

1 Introduction

Instruction Level Parallelism (ILP) offers the hope of greatly faster computers by automatically overlapping the execution of many machine-level instructions to complete tasks more quickly. An important class of ILP machine is the Very Long Instruction Word (VLIW) computer. These simple machines provide large numbers of execution resources, but require a sophisticated compiler to schedule the instructions.

A wide variety of scheduling techniques has been proposed, varying from simple basic block scheduling, to sophisticated global software pipelining [Gre00]. The engineering effort in implementing different schemes can vary enormously. A serious problem is that in many cases it is not known whether the benefits of a more sophisticated technique are sufficient to outweigh increased program development and maintenance time. Examples of this include the unclear benefits of DAG scheduling over single path scheduling [Mou94,PSM97], and trade-offs between the very many modulo scheduling algorithms that have been proposed.

Another important example is whether a global instruction scheduling algorithm should use tail duplication or compensation code. Code duplication is

* This work was supported by the Austrian Science Foundation (FWF) Project P13444-INF

R. Wilhelm (Ed.): CC 2001, LNCS 2027, pp. 200–212, 2001.

often necessary when moving operations across basic block boundaries during scheduling, and a compiler writer can choose between these two approaches. Tail duplication is easy to implement, but duplicates more code than necessary. Compensation code requires much more engineering effort, but offers the possibility of smaller code size and better instruction cache performance.

In this paper, we compare tail duplication [Mah96] and compensation code [Fis81] in the context of single path global instruction scheduling. Single path algorithms schedule the most likely path through a region of code as if it were one large basic block. We focus especially on Superblock Scheduling and Trace Scheduling, global scheduling algorithms that use tail duplication and compensation code respectively. We analyse both approaches, and present experimental results on relative performance in a highly optimising VLIW compiler.

The paper is organised as follows. In section 2 we examine the Trace Scheduling algorithm. Section 3 examines the role of tail duplication in Superblock Scheduling. In section 4 we compare the engineering effort required to implement the two approaches. Section 5 looks at the techniques from the viewpoint of the size of the resulting code. We examine situations where choosing either approach can affect the running time of the compiled program in section 6. Section 7 describes our experimental comparison of the two approaches. In section 8 we examine other work that has compared trade-offs in global scheduling. Finally, we draw conclusions in section 9.

2 Trace Scheduling

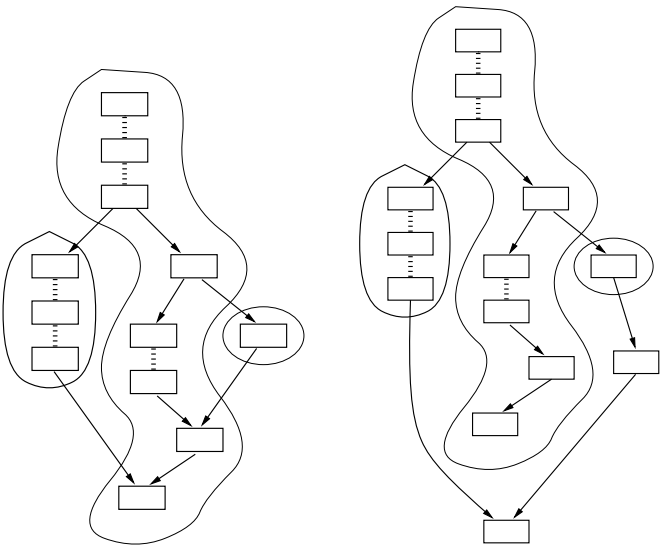
We will first describe global instruction scheduling in more detail. The simplest instruction schedulers use local scheduling, and group together independent operations in a single basic block. Global scheduling increases the available instruction level parallelism (ILP) by allowing operations to move across basic block boundaries. One of the earliest global scheduling algorithms is Trace Scheduling [Fis81,FERN84], which optimises the most commonly followed path in an acyclic region of code.

Trace scheduling works in a number of stages. First, traces are identified in the code. Figure 1(a) shows an example of trace selection. A trace is an acyclic sequence of basic blocks which are likely to be executed one after another, based on execution frequency statistics. Traces in an acyclic region are identified starting from the most frequently executed basic block which is not already part of a trace. This block is placed in a new trace, and the algorithm attempts to extend the trace backwards and forwards using the mutual most likely rule. Two blocks a and b, where b is a successor block of a, are mutual most likely if a is most frequently exited through the control flow edge from a to b, and b is most frequently entered through the same edge.

The trace selector continues to extend the trace until there is no mutual most likely block at either end of the trace, or where extending the trace would lead across a loop back-edge, or out of the acyclic region. An important feature of the mutual most likely rule is that at the start or end of any block there will

be at most one other block that satisfies the mutual most likely criteria. This means that each basic block will be in exactly one trace. This simplifies code transformations enormously.

Once the traces have been identified, they are scheduled, with the most frequent traces scheduled first. The scheduler removes the trace from the control flow graph and schedules it as the sequence of basic blocks were a single large basic block. This will re-order the sequence of operations to group together independent operations which can be executed simultaneously. The next step is to place the schedule back into the control flow graph (CFG).



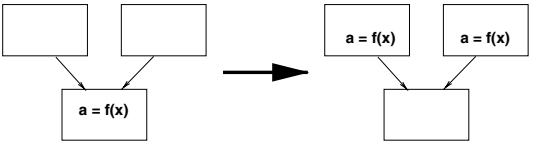
(a) Trace Selection

(b) Superblock Formation

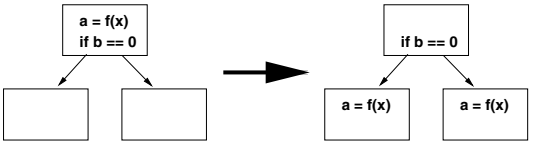
Fig. 1.

Placing the schedule back into the CFG is the most complicated part of Trace Scheduling. The problem is that moving an operation from one basic block to another can break the code. For example, consider the situation in figure 2(a). The operation *a =f(x)* appears after the control flow join in the original code, but scheduling results in the operation moving above the join. If, for example, the operation were to move into the left-hand block above the join, then if the join were reached during execution via the right-hand block, this operation would never be executed. To overcome this problem, *compensation code* is added at

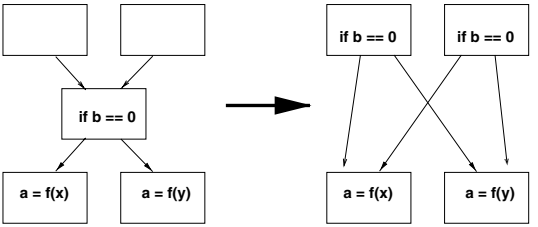
trace entry and exit points to ensure that all operations that should be executed are actually executed.



(a) Moving an operation above a join



(b) Moving an operation below a split



(c) Moving a branch above a join

Fig. 2. Compensation Code

Compensation code is simplest when moving non-branch operations across basic block boundaries. Figure 2(a) shows the effects of moving an operation above a control flow join. The operation must be duplicated and placed on every incoming path to the join. Similarly, figure 2(b) shows how operations may be moved below a control flow split (typically a conditional branch). The operation is copied to all outgoing paths of the split. The situation is considerably more complicated where branches move above joins or below below splits, and often results in significant changes in the shape of the control flow graph. Figure 2(c) shows a simple example of a branch moving above a join.

Generating compensation code is complicated. An operation may move past any combination of splits and/or joins. Merely deciding where to place duplicates

is far from simple. The situation can be even more complicated if renaming or other transformations must be applied at the same time. Modifying the shape of the control flow graph for moving branches across basic block boundaries is also complicated, especially where analysis information in the low level representation must be kept consistent. In order to avoid these engineering difficulties a new approach was taken when building the IMPACT compiler, perhaps the most developed research ILP compiler. In the next section, we describe this simpler approach.

3 Superblock Scheduling

Superblock scheduling [CCMmWH91,MCH+92] is similar to trace scheduling in that it attempts to optimise the most frequent path. A superblock is a trace that contains no control flow joins, except at the entry of the trace. Thus, after trace selection the algorithm removes joins from the trace by duplicating blocks which appear after the join. This is known as superblock formation, and an example is shown in figure 1(b). Wherever a block in the middle of a trace has more than one incoming control flow edge, that block is duplicated. It is important to note that duplicating this block will cause its successor blocks to have multiple predecessors, and these blocks must too be duplicated if they are in (but not the first block of) a trace.

After superblock formation, a number of optimisations are applied to each superblock, such as constant propagation, redundant load elimination and unrolling. Removing control flow joins often allows traditional optimisations to be more effective, since there is no need to take account of effects on rarely followed overlapping paths. In addition, Mahlke [Mah96] mentions that the IMPACT compiler also applies branch target expansion to increase the size of superblocks. This is, in effect, another round of tail duplication, as it copies code that appears after joins. In this paper, we do not consider any further duplication after superblock formation.

The next step is to schedule the operations in the superblock. The superblock scheduler is similar to, but simpler than, the scheme used by trace scheduling. The biggest difference is that since superblocks contain no joins, the scheduler does not move operations above joins. This greatly simplifies the code generation process. There is no need to duplicate operations or reshape the control flow graph for branches moving above joins. The superblock scheduler does, however, allow operations to move below branches. Thus, some compensation code may be need to be added after scheduling. But as described by Mahlke, Superblock Scheduling does not allow branches to be re-ordered, thus avoiding the most complicated part of compensation code for splits.

Superblock formation before scheduling allows almost all of the complexity for compensation code to be eliminated from the compiler. This greatly reduces the engineering effort in implementing and maintaining a global scheduling compiler. In the next section, we look at how great the difference can be.

4 Engineering Effort

There is no good way to compare the engineering difficulty of implementing different code duplication schemes. Any measure will be arbitrary and subjective to a degree. Nonetheless, we believe that some broad conclusions can be drawn. Perhaps the best measure of the complexity of compensation code is that Ellis devotes thirty eight pages of his PhD thesis [Ell85] to describing how to generate it in a Trace Scheduling compiler. The thesis also contains a further fifteen pages of discussion of possible improvements. The treatment is thorough, but not excessive. In contrast, Mahlke adequately describes superblock formation in less than two pages of his PhD thesis [Mah96]. Mahlke does not allow branches to be re-ordered, however, and only briefly describes operations moving below branches. Nonetheless, even if following a similar course would allow half or more of Ellis's material to be removed, his description of compensation code generation would still be very much larger than Mahlke's.

Another possible measure of relative complexity is our own implementation in the Chameleon compiler. Our code to perform tail duplication occupies about 400 lines of source code. It was possible to write this code in less than two weeks. Most of the time was devoted to understanding the Chameleon intermediate representation and debugging. Our code uses existing Chameleon functions for patching up the control flow graph when unrolling loops, however. If these functions were written from scratch, the code might be almost twice as large. In addition, the code is quite simple and straightforward.

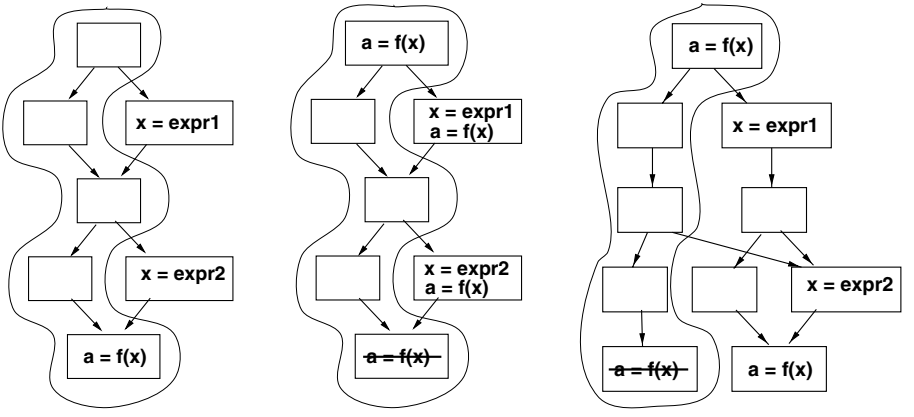
For scheduling, Chameleon implements the EPS++ [Mou97] software pipelining algorithm. This algorithm includes quite sophisticated compensation code generation, including compensation code for operations moving across loop iterations. It does not re-order branches, however, but it does include some other optimisations, such as redundancy elimination. After we adapted the EPS++ code for moving operations across basic block boundaries to Trace Scheduling, it occupied more than 4500 lines of code. Not all this code deals with compensation code generation, but even if only half of this code is really necessary for the compensation code in Trace Scheduling, it is still substantially larger than the code for superblock formation. In addition, this code is sometimes quite complicated, especially when moving branches above joins.

5 Code Growth

A simple characterisation of the difference between compensation code and tail duplication is that the former duplicates operations only where necessary, whereas the latter creates code growth whether it is needed or not. In general we would expect that tail duplication would produce substantially more code growth. In fact, this is not necessarily the case.

For an original region containing n operations, the original Trace Scheduling algorithm can create a final schedule containing approximately $O(n^n)$ operations [Ell85]. Most of this worst case code growth comes from reordering branches

which require large amounts of code duplication. A common strategy in global scheduling compilers is to disallow reordering of branches. This eliminates a complicated transformation from the compiler. In addition, the lost ILP from preventing some branches from being scheduled a few cycles earlier is probably low, since most branches are very predictable. This also reduces the worst case code growth from approximately $O(n^n)$ to $O(k^n)$, where k is the largest number of incoming control flow edges for any node in the CFG. The new worst case code growth corresponds to a situation where an acyclic graph is converted to a tree by repeatedly duplicating all nodes that have more than one predecessor. While still exponential, it is an enormous improvement. This is the strategy we follow in our implementation.



(a) Original (b) Trace Scheduling (c) Superblock Scheduling

Fig. 3. Comparison of code growth

Figure 3(b) shows an example where an operation is duplicated many times as it moves above a number of control flow joins. The operation moves up the control flow graph along the most likely path, and at each join a copy is added to the incoming off-path edge. Note that in many cases some or all of these duplicates can be eliminated [FGL94]. Many compilers, such as Chameleon and the Multiflow Trace compiler, add a redundancy elimination pass after compensation code generation which removes operations which recompute an existing value. In this example, however, the duplicates cannot be eliminated because the register x is redefined on each of the incoming paths.

The worst case code growth for tail duplication is substantially less. Recall that the trace selection phase results in each basic block in the acyclic region

being placed in exactly one trace. During superblock formation, incoming join edges that allow entry to the trace at any point other than the start are eliminated by duplicating the blocks in the trace which appear after the join. Given that each block is in only one trace, we can be sure that each block will be duplicated at most one time.

Figure 3(c) shows the effect of moving the same operation during Superblock Scheduling. Note that the operation `a = f(x)` is duplicated only once in the process of moving it from the last basic block to the first, whereas Trace Scheduling (figure 3(b)) requires that it is duplicated twice. Superblock formation never causes an operation to be duplicated more than once. Thus for an acyclic region containing n operations, after superblock formation the region will contain at most $2n$ operations.

It is important to bear in mind that there can be an enormous difference between average and worst case code growth. Global instruction scheduling algorithms similar to Trace Scheduling very rarely result in exponential code growth [Gre00,PSM97]. In most cases we would expect code size increases to be modest, and considerably less than that of tail duplication. Tail duplication, on the other hand, will often cause close to its worst case code growth, as it always duplicates blocks after joins whether or not the instructions following the join will be moved above it or not.

6 Speedup

In general, we would expect the speedup to be similar whether tail duplication or compensation code is used. Both approaches have advantages, however, that may result in better code in come circumstances. For example, tail duplication often makes other optimisations more effective, by allowing them to concentrate on a single superblock, regardless of definitions or usages on other control flow paths. For example Hwu et al [HMC+93] found that applying superblock formation made other traditional optimisations significantly more effective even without sophisticated scheduling[1].

Compensation code has an advantage over tail duplication when scheduling off-trace paths. Recall that in figure 3(b) we showed that Trace Scheduling can result in more compensation copies being generated. The two copies of operation `a = f(x)` are absorbed into other traces and scheduled as part of these traces. The situation with tail duplication and Superblock Scheduling is different. In figure 3(c) we see that the copy of `a = f(x)` remains in in a separate basic block after a join. This operation cannot be scheduled with the operations before the join, since superblocks cannot extend across a join. Thus, if an off-trace path is followed frequently, Superblock Scheduling may produce slower schedules.

[1] It appears that the baseline compiler used for comparison was rather poor. In many cases the reported speedups for Superblock Scheduling were more than four on a 4-issue processor, which is a strong sign that the baseline compiler is producing very poor code.

Code growth also has an important effect on speedup. Instruction cache misses can have a much greater effect on performance than a small difference in the length of the schedule of a rarely followed path. In general we expect that compensation code will produce less code growth, and so is likely to give better instruction cache performance.

7 Experimental Results

We implemented Trace Scheduling and Superblock Scheduling[2] in the Chameleon Compiler and ILP test-bed. Chameleon provides a highly optimizing ILP compiler for IBM's Tree VLIW architecture. The compiler performs many sophisticated traditional and ILP increasing optimizations. We compiled and scheduled several small benchmark programs whose inner loops contain branches, and four larger programs (*anagram*, *yacr-2*, *ks* and *bc*) with various inner loops. We ran the resulting programs on Chameleon's VLIW machine simulator.

The Chameleon compiler uses an existing C compiler (in our case *gcc*) as a front end to generate the original sequential machine code. The baseline for our speedup calculation is the number of cycles it takes to run this code on a simulated one ALU VLIW machine. This sequential code is then optimised to increase ILP, rescheduled by our global scheduling phase, and registers are reallocated. We run this scheduled code on a simulated machine which can execute four ALU operations per cycle. The speedup is the baseline cycles, divided by the number needed by the 4-ALU VLIW.

Benchmark	Description
eight	Solve eight queens problem
wc	Unix word count utility
bubble	Bubble sort array of random integers
binsearch	Binary search sorted array
tree	Tree sort array of random integers
branch	Inner loop branch test program
eqn	Inner loop of eqntott
anagram	Generate anagrams of strings
ks	Graph partitioning tool
yacr-2	Channel routiner
bc	GNU bc calculator

Figure 4 shows the speedup achieved by both algorithms on the benchmarks. As expected, the performance of the two algorithms is very similar, because both concentrate on scheduling the most common path. Where branches are

[2] For control purposes, we also tested applying Trace Scheduling after superblock formation, and Superblock Scheduling without superblock formation. The results didn't contain any surprises, and are thus not presented here.

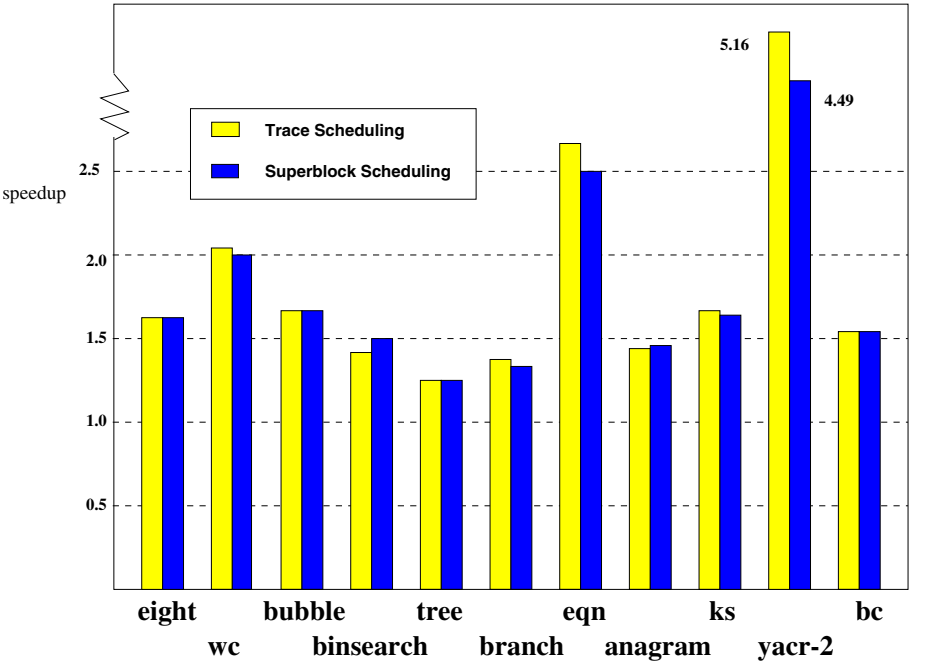


Fig. 4. Speedup

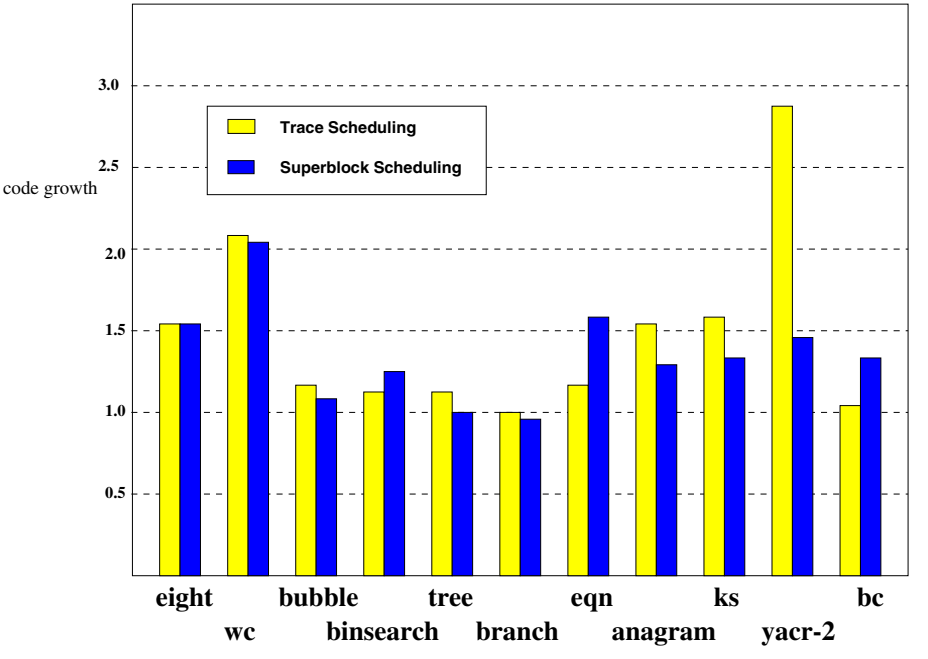


Fig. 5. Code Growth

very unpredictable, Trace Scheduling sometimes does a little better (in *eqn* for example) for reasons outlined in section 6. One remarkable result is for *yacr-2*, where the speedup is greater than the width of the VLIW. In this case, the benefit from other optimisations applied by the VLIW compiler are greater than the gain from global instruction scheduling. In effect, the VLIW optimiser is making up for poor optimisation by the front-end compiler. On investigation, we found that some of the code size increasing optimisations that the VLIW compiler makes are applied to a lesser degree if the original code size is larger. Thus, superblock formation before optimisation can cause these optimisations to be applied less. This is the main reason that Trace Scheduling outperforms Superblock scheduling on *yacr-2*.

The code growth results shown in figure 5 are much more interesting. As expected, Trace Scheduling produces less code growth than Superblock Scheduling for *binsearch* and *eqn* and *bc*. In the case of *eight* the code growth is identical — almost all operations are moved above the joins, so there is no difference between the two approaches. What is surprising is that Trace Scheduling produces more code growth for many benchmarks. In the case of *wc* this arises from its complicated control flow in the inner loop, which causes close to worst case code growth similar to that in figure 3(b). Both *anagram* and *ks* contain similarly complicated control flow, although not in their most common inner loops. Again for *yacr-2* we found that other code size increasing optimisations are primarily responsible for the code growth. There is one particulatly large function with complex control flow in *yacr-2* which causes more than average code growth with compensation code. But the great majority of the difference is accounted for by other optimisations and the result shown does not tell us much useful about compensation code or tail duplication.

Looking at the speedup and code growth figures together, it is possible to reach more general conclusions. Trace Scheduling is sometimes a little better and sometimes a little worse than Superblock Scheduling. Contrary to what we expected, compensation code does not produce consistently less code growth than tail duplication. In fact, it often produces more code growth. Given the considerably greater engineering work in implementing compensation code, our results suggest that unless it can be improved in the future, it is not worth the effort.

8 Related Work

Most work in global scheduling concentrates on proposing new schemes, but a number of authors have compared their scheme with an existing one. Moudgill [Mou94] examined a number of different acyclic scheduling regions such as DAG and single path for Fortran programs. In most cases the benefit was small. Havanki et al [WHC98] compared tree scheduling with Superblock Scheduling and reported some speedups for integer code. Mahlke [Mah96] measured the performance improvement of predicated code over Superblock Scheduling. Perhaps the most developed work on comparing strategies is [PSM97] which looks at a

number of issues such as memory disambiguation, probability guided scheduling, single path versus multipath, and software pipelining versus loop unrolling. We are not aware of any existing work which looks at the trade-offs between tail duplication and compensation code.

9 Conclusions

We have examined the problem of maintaining the correctness of code when moving operations above control flow joins during single path global instruction scheduling. Two techniques for doing this are generating compensation code and tail duplication. Compensation code is more complicated and difficult to implement, but offers the possibility of less code growth. We implemented two global scheduling algorithms which use the two techniques in a highly optimising VLIW compiler. Experimental results show that compensation code does not always create less code growth and often creates more. Given the much greater engineering effort in implementing compensation code, it is unlikely to be worth the effort, unless improvements can be made in the future.

Acknowledgments. We would like to thank the VLIW group at IBM's T. J. Watson Research Center for providing us with the Chameleon experimental test-bed. Special thanks to Mayan Moudgill and Michael Gschwind. We are also grateful to Sylvain Lelait for his comments on an earlier version of this work.

References

CCMmWH91. Pohua P. Chang, William Y. Chen, Scott A. Mahlke, and Wen mei W. Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275. IEEE, May 1991.

Ell85. John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1985.

FERN84. Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. In *ACM '84 Symposium on Compiler Construction*, pages 37–47. ACM, June 1984.

FGL94. Stefan M. Freudenberger, Thomas R. Gross, and P. Geoffrey Lowney. Avoidance and suppression of compensating code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems*, 16(4):1156–1214, 1994.

Fis81. Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

Gre00. David Gregg. Global software pipelining with iteration preselection. In *CC–2000 International Conference on Compiler Construction*, LNCS 1781, March 2000.

HMC+93. Wen-mei Hwu, Scott Mahlke, William Chen, Pohua Chang, Nancy Warter, Roger Bringmann, Roland Oullette, Richard Hank, Tokuzo Kiyohara, Grant Haab, John Holm, and Daniel Lavery. The superblock: An effective tachnique for VLIW and superscalar compilation. *The Journal of Supercomputing*, pages 229–248, 1993.

Mah96. Scott Alan Mahlke. *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches*. PhD thesis, Department of Elecrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1996.

MCH+92. Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247. ACM, September 1992.

Mou94. Mayan Moudgill. *Implementing and Exploiting Static Speculation and Multiple Issue Processors*. PhD thesis, Cornell University, May 1994.

Mou97. Mayan Moudgill. Source code of the chameleon compiler. The source code is the only written description of the EPS++ algorithm, 1997.

PSM97. S. Park, S. Shim, and S. Moon. Evaluation of scheduling techniques on a sparc-based VLIW testbed. In *30th International Symposium on Microarchitecture*. IEEE, November 1997.

WHC98. S. Banerjia W. Havanki and T. Conte. Treegion scheduling for wide-issue processors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.